EXHIBIT B

4.9 Verification of class Files

Even though Sun's Java compiler attempts to produce only class files that satisfy all the static constraints in the previous sections, the Java Virtual Machine has no guarantee that any file it is asked to load was generated by that compiler, or is properly formed. Applications such as Sun's HotJava World Wide Web browser do not download source code which they then compile; these applications download already- compiled `class` files. The HotJava browser needs to determine whether the `class` file was produced by a trustworthy Java compiler or by an adversary attempting to exploit the interpreter.

An additional problem with compile-time checking is version skew. A user may have successfully compiled a class, say `PurchaseStockOptions`, to be a subclass of `TradingClass`. But the definition of `TradingClass` might have changed in a way that is not compatible with preexisting binaries since the time the class was compiled. Methods might have been deleted, or had their return types or modifiers changed. Fields might have changed types or changed from instance variables to class variables. The access modifiers of a method or variable may have changed from `public` to `private`. For a discussion of these issues, see Chapter 13, "Binary Compatibility," in *The Java Language Specification*.

Because of these potential problems, the Java Virtual Machine needs to verify for itself that the desired constraints hold on the `class` files it attempts to incorporate. A well-written Java Virtual Machine emulator could reject poorly formed instructions when a `class` file is loaded. Other constraints could be checked at run time. For example, a Java Virtual Machine implementation could tag runtime data and have each instruction check that its operands are of the right type.

Instead, Sun's Java Virtual Machine implementation verifies that each `class` file it considers untrustworthy satisfies the necessary constraints at linking time (§2.16.3). Structural constraints on the Java Virtual Machine code are checked using a simple theorem prover.

Linking-time verification enhances the performance of the interpreter. Expensive checks that would otherwise have to be performed to verify constraints at run time for each interpreted instruction can be eliminated. The Java Virtual Machine can assume that these checks have already been performed. For example, the Java Virtual Machine will already know the following:

- There are no operand stack overflows or underflows.
- All local variable uses and stores are valid.
- The arguments to all the Java Virtual Machine instructions are of valid types.

Sun's `class` file verifier is independent of any Java compiler. It should certify all code generated by Sun's current Java compiler; it should also certify code that other compilers can generate, as well as code that the current compiler could not possibly generate. Any `class` file that satisfies the structural criteria and static constraints will be certified by the verifier.

The `class` file verifier is also independent of the Java language. Other languages can be compiled into the `class` format, but will only pass verification if they satisfy the same constraints as a `class` file compiled from Java source.

4.9.1 The Verification Process

The `class` file verifier operates in four passes:

Pass 1: When a prospective `class` file is loaded (§2.16.2) by the Java Virtual Machine, the Java Virtual Machine first ensures that the file has the basic format of a Java `class` file. The first four bytes must contain the right magic number. All recognized attributes must be of the proper length. The `class` file must not be truncated or have extra bytes at the end. The constant pool must not contain any superficially unrecognizable information.

While `class` file verification properly occurs during class linking (§2.16.3), this check for basic `class` file integrity is necessary for any interpretation of the `class` file contents and can be considered to be logically part of the verification process.

Pass 2: When the `class` file is linked, the verifier performs all additional verification that can be done without looking at the `code` array of the `Code` attribute (§4.7.4). The checks performed by this pass include the following:

- Ensuring that `final` classes are not subclassed, and that `final` methods are not overridden.
- Checking that every class (except `Object`) has a superclass.
- Ensuring that the constant pool satisfies the documented static constraints; for example, class references in the constant pool must contain a field that points to a `CONSTANT_Utf8` string reference in the constant pool.
- Checking that all field references and method references in the constant pool have valid names, valid classes, and a valid type descriptor.

Note that when it looks at field and method references, this pass does not check to make sure that the given field or method actually exists in the given class; nor does it check that the type descriptors given refer to real classes. It only checks that these items are well formed. More detailed checking is delayed until passes 3 and 4.

Pass 3: Still during linking, the verifier checks the `code` array of the `Code` attribute for each method of the `class` file by performing data-flow analysis on each method. The verifier ensures that at any given point in the program, no matter what code path is taken to reach that point:

- The operand stack is always the same size and contains the same types of objects.
- No local variable is accessed unless it is known to contain a value of an appropriate type.
- Methods are invoked with the appropriate arguments.
- Fields are assigned only using values of appropriate types.
- All opcodes have appropriate type arguments on the operand stack and in the local variables.

For further information on this pass, see Section 4.9.2, "The Bytecode Verifier."

Pass 4: For efficiency reasons, certain tests that could in principle be performed in Pass 3 are delayed until the first time the code for the method is actually invoked. In so doing, Pass 3 of the verifier avoids loading `class` files unless it has to.

For example, if a method invokes another method that returns an instance of class `A`, and that instance is only assigned to a field of the same type, the verifier does not bother to check if the class `A` actually exists. However, if it is assigned to a field of the type `B`, the definitions of both `A` and `B` must be loaded in to ensure that `A` is a subclass of `B`.

Pass 4 is a virtual pass whose checking is done by the appropriate Java Virtual Machine instructions. The first time an instruction that references a type is executed, the executing instruction does the following:

- Loads in the definition of the referenced type if it has not already been loaded.
- Checks that the currently executing type is allowed to reference the type.
- Initializes the class, if this has not already been done.

The first time an instruction invokes a method, or accesses or modifies a field, the executing instruction does the following:

- Ensures that the referenced method or field exists in the given class.
- Checks that the referenced method or field has the indicated descriptor.
- Checks that the currently executing method has access to the referenced method or field.

The Java Virtual Machine does not have to check the type of the object on the operand stack. That check has already been done by Pass 3. Errors that are detected in Pass 4 cause instances of subclasses of `LinkageError` to be thrown.

A Java Virtual Machine is allowed to perform any or all of the Pass 4 steps, except for class or interface initialization, as part of Pass 3; see 2.16.1, "Virtual Machine Start-up" for an example and more discussion.

In Sun's Java Virtual Machine implementation, after the verification has been performed, the instruction in the Java Virtual Machine code is replaced with an alternative form of the instruction (see Chapter 9, "An Optimization"). For example, the opcode `new` is replaced with `new_quick`. This alternative instruction indicates that the verification needed by this instruction has taken place and does not need to be performed again. Subsequent invocations of the method will thus be faster. It is illegal for these alternative instruction forms to appear in `class` files, and they should never be encountered by the verifier.

4.9.2 The Bytecode Verifier

As indicated earlier, Pass 3 of the verification process is the most complex of the four passes of `class` file verification. This section looks at the verification of Java Virtual Machine code in more detail.

The code for each method is verified independently. First, the bytes that make up the code are broken up into a sequence of instructions, and the index into the `code` array of the start of each instruction is placed in an array. The verifier then goes through the code a second time and parses the instructions. During this pass a data structure is built to hold information about each Java Virtual Machine instruction in the method. The operands, if any, of each instruction are checked to make sure they are valid. For instance:

- Branches must be within the bounds of the `code` array for the method.
- The targets of all control-flow instructions are each the start of an instruction. In the case of a *wide* instruction, the *wide* opcode is considered the start of the instruction, and the opcode giving the operation modified by that *wide* instruction is not considered to start an instruction. Branches into the middle of an instruction are disallowed.
- No instruction can access or modify a local variable at an index greater than the number of local variables that its method indicates it uses.
- All references to the constant pool must be to an entry of the appropriate type. For example: the instruction *ldc* can only be used for data of type `int` or `float`, or for instances of class `String`; the instruction *getfield* must reference a field.
- The code does not end in the middle of an instruction.
- Execution cannot fall off the end of the code.
- For each exception handler, the starting and ending point of code protected by the handler must be at the beginning of an instruction. The starting point must be before the ending point. The exception handler code must start at a valid instruction, and it may not start at an opcode being modified by the *wide* instruction.

For each instruction of the method, the verifier records the contents of the operand stack and the contents of the local variables prior to the execution of that instruction. For the operand stack, it needs to know the stack height and the type of each value on it. For each local variable, it needs to know either the type of the contents of that local variable, or that the local variable contains an unusable or unknown value (it might be uninitialized). The bytecode verifier does not need to distinguish between the integral types (e.g., `byte`, `short`, `char`) when determining the value types on the operand stack.

Next, a data-flow analyzer is initialized. For the first instruction of the method, the local variables which represent parameters initially contain values of the types indicated by the method's type descriptor; the operand stack is empty. All other local variables contain an illegal value. For the other instructions, which have not been examined yet, no information is available regarding the operand stack or local variables.

Finally, the data-flow analyzer is run. For each instruction, a "changed" bit indicates whether this instruction needs to be looked at. Initially, the "changed" bit is only set for the first instruction. The data-flow analyzer executes the following loop:

1. Select a virtual machine instruction whose "changed" bit is set. If no instruction remains whose "changed" bit is set, the method has successfully been verified. Otherwise, turn off the "changed" bit of the selected instruction.
2. Model the effect of the instruction on the operand stack and local variables:
   - If the instruction uses values from the operand stack, ensure that there are a sufficient number of values on the stack and that the top values on the stack are of an appropriate type. Otherwise, verification fails.
   - If the instruction uses a local variable, ensure that the specified local variable contains a value of the appropriate type. Otherwise, verification fails.
   - If the instruction pushes values onto the operand stack, ensure that there is sufficient room on the operand stack for the new values. Add the indicated types to the top of the modeled operand stack.
   - If the instruction modifies a local variable, record that the local variable now contains the new type.
3. Determine the instructions that can follow the current instruction. Successor instructions can be one of the following:
   - The next instruction, if the current instruction is not an unconditional control transfer instruction (for instance *goto*, *return* or *athrow*). Verification fails if it is possible to "fall off" the last instruction of the method.
   - The target(s) of a conditional or unconditional branch or switch.
   - Any exception handlers for this instruction.
4. Merge the state of the operand stack and local variables at the end of the execution of the current instruction into each of the successor instructions. In the special case of control transfer to an exception handler, the operand stack is set to contain a single object of the exception type indicated by the exception handler information.
   - If this is the first time the successor instruction has been visited, record that the operand stack and local variables values calculated in steps 2 and 3 are the state of the operand stack and local variables prior to executing the successor instruction. Set the "changed" bit for the successor instruction.
   - If the successor instruction has been seen before, merge the operand stack and local variable values calculated in steps 2 and 3 into the values already there. Set the "changed" bit if there is any modification to the values.
5. Continue at step 1.

To merge two operand stacks, the number of values on each stack must be identical. The types of values on the stacks must also be identical, except that differently typed `reference` values may appear at corresponding places on the two stacks. In this case, the merged operand stack contains a `reference` to an instance of the first common superclass or common superinterface of the two types. Such a reference type always exists because the type `Object` is a supertype of all class and interface types. If the operand stacks cannot be merged, verification of the method fails.

To merge two local variable states, corresponding pairs of local variables are compared. If the two types are not identical, then unless both contain `reference` values, the verifier records that the local variable contains an unusable value. If both of the pair of local variables contain `reference` values, the merged state contains a `reference` to an instance of the first common superclass of the two types.

If the data-flow analyzer runs on a method without reporting a verification failure, then the method has been successfully verified by Pass 3 of the `class` file verifier.

Certain instructions and data types complicate the data-flow analyzer. We now examine each of these in more detail.

4.9.3 Long Integers and Doubles

Values of the `long` and `double` types each take two consecutive words on the operand stack and in the local variables.

Whenever a `long` or `double` is moved into a local variable, the subsequent local variable is marked as containing the second half of a `long` or `double`. This special value indicates that all references to the `long` or `double` must be through the index of the lower-numbered local variable.

Whenever any value is moved to a local variable, the preceding local variable is examined to see if it contains the first word of a `long` or a `double`. If so, that preceding local variable is changed to indicate that it now contains an unusable value. Since half of the `long` or `double` has been overwritten, the other half must no longer be used.

Dealing with 64-bit quantities on the operand stack is simpler; the verifier treats them as single units on the stack. For example, the verification code for the *dadd* opcode (add two `double` values) checks that the top two items on the stack are both of type `double`. When calculating operand stack length, values of type `long` and `double` have length two.

Untyped instructions that manipulate the operand stack must treat values of type `double` and `long` as atomic. For example, the verifier reports a failure if the top value on the stack is a `double` and it encounters an instruction such as *pop* or *dup*. The instructions *pop2* or *dup2* must be used instead.

4.9.4 Instance Initialization Methods and Newly Created Objects

Creating a new class instance is a multistep process. The Java statement

```
...

new myClass(i, j, k);
...
```

can be implemented by the following:

```
...

new         #1              // Allocate uninitialized space for myClass
dup                 // Duplicate object on the operand stack
iload_1                     // Push i
iload_2                     // Push j
iload_3                     // Push k
invokespecial myClass.<init>                // Initialize object
...
```

This instruction sequence leaves the newly created and initialized object on top of the operand stack. (More examples of compiling Java code to the instruction set of the Java '

The instance initialization method `<init>` for class `myClass` sees the new uninitialized object as its `this` argument in local variable `0`. It must either invoke an alternative instance initialization method for class `myClass` or invoke the initialization method of a superclass on the `this` object before it is allowed to do anything else with `this`.

When doing dataflow analysis on instance methods, the verifier initializes local variable `0` to contain an object of the current class, or, for instance initialization methods, local variable `0` contains a special type indicating an uninitialized object. After an appropriate initialization method is invoked (from the current class or the current superclass) on this object, all occurrences of this special type on the verifier's model of the operand stack and in the local variables are replaced by the current class type. The verifier rejects code that uses the new object before it has been initialized or that initializes the object twice. In addition, it ensures that every normal return of the method has either invoked an initialization method in the class of this method or in the direct superclass.

Similarly, a special type is created and pushed on the verifier's model of the operand stack as the result of the Java Virtual Machine instruction *new*. The special type indicates the instruction by which the class instance was created and the type of the uninitialized class instance created. When an initialization method is invoked on that class instance, all occurrences of the special type are replaced by the intended type of the class instance. This change in type may propagate to subsequent instructions as the dataflow analysis proceeds.

The instruction number needs to be stored as part of the special type, as there may be multiple not-yet-initialized instances of a class in existence on the operand stack at one time. For example, the Java Virtual Machine instruction sequence that implements

```
new InputStream(new Foo(), new InputStream("foo"))
```

may have two uninitialized instances of `InputStream` on the operand stack at once. When an initialization method is invoked on a class instance, only those occurrences of the special type on the operand stack or in the registers that are the *same object* as the class instance are replaced.

A valid instruction sequence must not have an uninitialized object on the operand stack or in a local variable during a backwards branch, or in a local variable in code protected by an exception handler or a `finally` clause. Otherwise, a devious piece of code might fool the verifier into thinking it had initialized a class instance when it had, in fact, initialized a class instance created in a previous pass through the loop.

4.9.5 Exception Handlers

Java Virtual Machine code produced from Sun's Java compiler always generates exception handlers such that:

- The ranges of instructions protected by two different exception handlers always are either completely disjoint, or else one is a subrange of the other. There is never a partial overlap of ranges.
- The handler for an exception will never be inside the code that is being protected.
- The only entry to an exception handler is through an exception. It is impossible to fall through or "goto" the exception handler.

These restrictions are not enforced by the `class` file verifier since they do not pose a threat to the integrity of the Java Virtual Machine. As long as every nonexceptional path to the exception handler causes there to be a single object on the operand stack, and as long as all other criteria of the verifier are met, the verifier will pass the code.

4.9.6 Exceptions and `finally`

Given the fragment of Java code

```
...
```

```
try {
    startFaucet();
    waterLawn();
} finally {
    stopFaucet();
}
...
```

the Java language guarantees that `stopFaucet` is invoked (the faucet is turned off) whether we finish watering the lawn or whether an exception occurs while starting the fauc

To implement the `try-finally` construct, the Java compiler uses the exception-handling facilities together with two special instructions *jsr* ("jump to subroutine") and *ret* ("return from subroutine"). The `finally` clause is compiled as a subroutine within the Java Virtual Machine code for its method, much like the code for an exception handler. When a *jsr* instruction that invokes the subroutine is executed, it pushes its return address, the address of the instruction after the *jsr* that is being executed, onto the operand stack as a value of type `returnAddress`. The code for the subroutine stores the return address in a local variable. At the end of the subroutine, a *ret* instruction fetches the return address from the local variable and transfers control to the instruction at the return address.

Control can be transferred to the `finally` clause (the `finally` subroutine can be invoked) in several different ways. If the `try` clause completes normally, the `finally` subroutine is invoked via a *jsr* instruction before evaluating the next Java expression. A `break` or `continue` inside the `try` clause that transfers control outside the `try` clause executes a *jsr* to the code for the `finally` clause first. If the `try` clause executes a `return`, the compiled code does the following:

1. Saves the return value (if any) in a local variable.
2. Executes a *jsr* to the code for the `finally` clause.
3. Upon return from the `finally` clause, returns the value saved in the local variable.

The compiler sets up a special exception handler which catches any exception thrown by the `try` clause. If an exception is thrown in the `try` clause, this exception handler does the following:

1. Saves the exception in a local variable.
2. Executes a *jsr* to the `finally` clause.
3. Upon return from the `finally` clause, rethrows the exception.

For more information about the implementation of Java's `try-finally` construct, see Section 7.13, "Compiling finally."

The code for the `finally` clause presents a special problem to the verifier. Usually, if a particular instruction can be reached via multiple paths and a particular local variable contains incompatible values through those multiple paths, then the local variable becomes unusable. However, a `finally` clause might be called from several different places, yielding several different circumstances:

- The invocation from the exception handler may have a certain local variable that contains an exception.
- The invocation to implement `return` may have some local variable that contains the return value.
- The invocation from the bottom of the `try` clause may have an indeterminate value in that same local variable.

The code for the `finally` clause itself might pass verification, but after updating all the successors of the *ret* instruction, the verifier would note that the local variable that the exception handler expects to hold an exception, or that the return code expects to hold a return value, now contains an indeterminate value.

Verifying code that contains a `finally` clause is complicated. The basic idea is the following:

- Each instruction keeps track of the list of *jsr* targets needed to reach that instruction. For most code, this list is empty. For instructions inside code for the `finally` clause, it is of length one. For multiply nested `finally` code (extremely rare!), it may be longer than one.
- For each instruction and each *jsr* needed to reach that instruction, a bit vector is maintained of all local variables accessed or modified since the execution of the *jsr* instruction.
- When executing the *ret* instruction, which implements a return from a subroutine, there must be only one possible subroutine from which the instruction can be returning. Two different subroutines cannot "merge" their execution to a single *ret* instruction.
- To perform the data-flow analysis on a *ret* instruction, a special procedure is used. Since the verifier knows the subroutine from which the instruction must be returning, it can find all the *jsr* instructions that call the subroutine and merge the state of the operand stack and local variables at the time of the *ret* instruction into the operand stack and local variables of the instructions following the *jsr*. Merging uses a special set of values for the local variables:
- For any local variable for which the bit vector (constructed above) indicates that the subroutine has accessed or modified, use the type of the local variable at the time of the *ret*.
- For other local variables, use the type of the local variable before the *jsr* instruction.